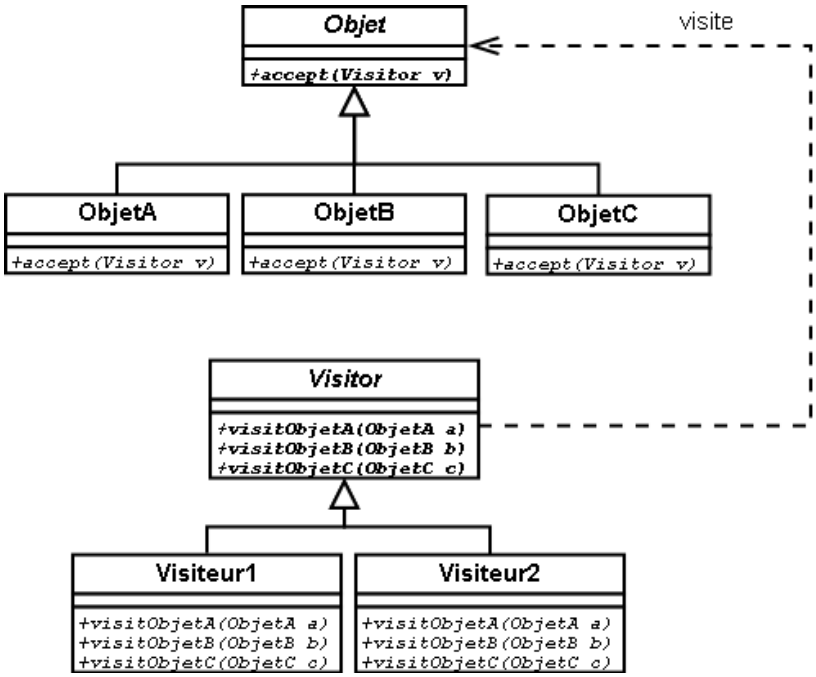
Application du Pattern Visitor

A travers les exercices de ce TP, nous allons voir le design Pattern Visitor et son application.

Le pattern Visitor permet de séparer de manière très simple les données et les traitements associés. Ci-dessous le diagramme de classe simplifié du Visitor. Le diagramme présente 2 hiérarchies :

- *la hiérarchie des objets qui est le support de données*
- *la hiérarchie des Visiteurs (contenant les traitements sur les données)*



Le design Pattern « visitor » est représenté par une classe « visiteur » d'un côté, et une classe quelconque qui sera « visitée » de l'autre côté. Le visiteur a pour fonction le traitement de certains types d'objets. (ici : objetA, objetB, objetC). Selon l'objet qu'il visite, il effectuera un traitement différent. (visitObjetA(), visiteObjetB()…) il est ainsi possible de créer différents types de Visitor (Visitor1, Visitor2) pour différents comportements.

La classe visitée doit définir une méthode accept() qui sera implémentée par chacune des sous-classes. Cette méthode est en charge d'appeler la méthode de la classe Visitor dont le type de l'argument en entrée de la méthode correspond au type de classe traitée. Ainsi, l'objet A appellera la méthode visitObjetA() du visitor et ainsi de suite…

Exercice 1 :

1°) Créer l'interface *IVisitor* définissant les méthodes virtuelles pures suivantes :

- *void visit(Ship* s)*
- *void visit(Hull* h)*
- *void visit(Engine*)*

2°) Créer l'interface *IVisited* définissant la méthode virtuelle pure *void accept(IVisitor* v)*.

3°) Modifier les classes *Ship*, *Hull et Engine* afin qu'elles implémentent la méthode *accept()*.

Cette méthode, relativement simple, permet d'appeler l'une des méthodes surchargées de l'interface définie en 1).

Selon le type d'objet visité, son implémentation peut changer mais la plupart du temps, l'algorithme associé au *pattern* se réduit à une instruction:

```
void Ship::accept(IVisitor* visitor)
{
  Visitor->visit(this) ;
}
```

Implémenter cette méthode dans chacune des classes.

4°) Dans la classe *Ship,* définir et déclarer la méthode suivante :

- *std::list<IVisited*> getVisitedItem()*.

Cette méthode doit retourner une liste de tous les objets agrégés à la classe Ship (i.e : déclarés et instanciés dans la classe *Ship*) qui implémentent l'interface *IVisited*.

5°) Créer une classe *XmlVisitor* qui implémente l'interface *IVisitor* :

Cette classe crée un fichier texte unique au format XML. Chaque appel aux méthodes de cette interface doit écrire dans le format spécifié les valeurs des attributs définis dans l'objet passé en paramètre de la méthode, soit pour exemple :

Classe	***Hull***	***Engine***	***Ship***
Traitement à réaliser dans la classe XmlVisitor	*<Hull>* *<Solidity></Solidity>* *</Hull>*	*<Engine>* *<Speed></Speed>* *</Engine>*	*<Ship>* *…. ?.....* *</Ship>*

N.B : Faites en sorte que la méthode *void visit(Ship* s)* fasse un appel récursif aux autres méthodes, en vous aidant notamment de la méthode définie en 4).

6°) Ajouter après la création d'un objet de type *Ship* les lignes de code suivantes :

```
{
  Ship* s = createShip(…) ;
  IVisitor* v = new XmlVisitor() ;
  s->accept(v);
}
```

Observer le résultat. Vérifier la cohérence du fichier obtenu en sortie.

Exercice 2

1°) Ajouter à la classe définie dans l'exercice 3 du TP1 une nouvelle méthode *void cycleOut()* qui implémentera l'algorithme suivant:

- La méthode parcourt la position de chacun des quais.
- Si un quai est occupé, alors l'algorithme déplace l'instance de la classe *Ship* choisie d'une case dans la matrice en direction de la sortie. (même règle de déplacement que pour l'entrée). A la prochaine itération, elle continuera le déplacement jusqu'à destination.
- si le quai est inoccupé, elle passe au quai suivant.

2°) Appeler les méthodes *cycle* et *cyleOut* successivement et de façon itérative (dans une boucle) afin de simuler un trafic entrant et sortant au sein de votre application.

N.B : Ne pas oublier de gérer les éventuels cas de collision.